VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Procedurální generování terénu

Procedural Terrain Generation

2014 Jakub Rak

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jakub Rak**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Procedurální generování terénu
Procedural Terrain Generation

Zásady pro vypracování:

Dnešní moderní grafické karty díky svému výkonu nabízejí při vykreslování velké možnosti. Cílem této práce je srovnat algoritmy pro generování a vykreslování terénu s přihlédnutím na výpočetní a paměťové možnosti dnešních grafických karet.

1. Nastudujte algoritmy pro generování terénu.
2. Zaměřte se zejména na algoritmy vhodné pro moderní grafické karty s podporou OpenGL 4.x.
3. Jednotlivé algoritmy teoreticky popište a charakterizujte jejich použití, výhody a nevýhody.
4. Vytvořte demonstrační aplikace pro vybrané algoritmy, které budou umožňovat jejich srovnávání (výpočetní náročnost, paměťová náročnost, časová náročnost apod.).

Seznam doporučené odborné literatury:

[1] D. Shreiner, G. Sellers, J.M. Kessenich and B. M. Licea-Kane, OpenGL Programming Guide, 8th Edition. 2013. 984 p. ISBN 978-0-321-77303-6
[2] http://www.opengl.org/

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Martin Němec, Ph.D.**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 2014 ..... Jakub Dak .....

Tímto bych rád poděkoval svému vedoucímu, Ing. Martinu Němcovi, Ph.D., za poskytnuté odborné rady, za neocenitelné zkušenosti a za všechen čas, jenž mi takto věnoval.

Abstrakt

Používání procedurálních technik se v dnešní době stále rozšiřuje, zejména díky stále rychlejšímu hardwaru. Výhody procedurálních technik jsou zejména vysoká úroveň abstrakce a algoritmus schopný vygenerovat data jenž zabírají více místa než on sám. Tato páce se zaměřuje na popis některých algoritmů používaných pro generaci procedurálního terénu. Následuje popis demonstrační aplikace a začátky její optimalizace.

Klíčová slova: procedurální, terén, generování, vizualizace, OpenGL, GLSL, optimalizace, Perlin noise, Worley noise

Abstract

Usage of procedural generation techniques is steadily increasing, mainly thanks to increasingly powerful hardware. Advantages of procedural techniques are especially high level of abstraction and algorithm capable of generating data much bigger than itself. This thesis aims to describe some of the algorithms used to procedurally generate terrain. Follows a description of the demonstration application and beginnings of its optimization.

Keywords: procedural, terrain, generation, visualization, OpenGL, GLSL, optimization, Perlin noise, Worley noise

Seznam použitých zkratek a symbolů

CPU	–	central processing unit – procesor, vykonává instrukce a aritmetické operace
FPS	–	frames per second – počet snímků za sekundu
HDD	–	hard disk drive – pevný disk, určen pro dlouhodobé bezpříkonové ukládání dat
Mesh	–	povrch tělesa – obyvkle reprezentován body, trojúhelníky a někdy indiciemi
Frustum	–	Uříznutá pyramida reprezentující pohled kamery
RAM	–	random–access memory – hlavní paměť počítače, slouží pro ukládání dat
GPU	–	graphics processing unit – grafická karta, má svůj vlastní procesor i RAM, pro zjednodušení budeme obojí nazývat GPU
OS	–	Operační systém
Realtime	–	Vykreslování v reálném čase – snaha vykreslovat co nejrychleji

Obsah

1 Úvod **2**

2 Reprezentace Terénu **3**
2.1 Mesh . . . 3
2.1.1 Posílání meshe před každým snímkem . . . 3
2.1.2 Poslání meshe jednou, Vertex Buffer Object . . . 4
2.2 Výšková mapa . . . 4
2.3 Objemová reprezentace . . . 5

3 Procedurální generování **6**
3.1 Výhody . . . 6
3.2 Nevýhody . . . 6
3.3 Algoritmy . . . 7
3.3.1 Perlin noise . . . 7
3.3.2 Simplex noise . . . 9
3.3.3 Worley noise . . . 10
3.3.4 Diamond-square algoritmus . . . 12

4 Popis realizace demonstrační aplikace **14**
4.1 Použité technologie . . . 14
4.1.1 Tessellace . . . 14
4.1.2 Triplanar texture projection . . . 15
4.1.3 Component a Transform design . . . 16
4.1.4 Detail textures . . . 17
4.2 Optimalizace . . . 18
4.2.1 View frustum culling . . . 18
4.2.2 Instanced rendering . . . 20
4.2.3 Porovnání konteineru vectoru a listu . . . 21
4.2.4 Porovnání procházení pole pomocí iterátoru a pointeru . . . 22
4.2.5 Vykreslovací smyčka . . . 22
4.2.6 Level of Detail . . . 23
4.2.7 Více vláknová aplikace . . . 23
4.3 Nepřesnost čísel s plovoucí desetinnou čárkou . . . 24

5 Zhodnocení **25**

6 Závěr **27**

7 Reference **28**

Přílohy **29**

A Příloha **30**

1 Úvod

Procedurální techniky jsou teoreticky používány již od počátku počítačů. Jsme schopni vygenerovat požadované informace, které zabírají mnohem více místa než sám algoritmus. Tyto techniky jsou velice silným nástrojem, jenž usnadňuje práci v širokém spektru profesí. Existuje řada vysoce specializovaných softwarů (Terragen [14], E-On Vue [15], World Machine [16], atd...), které se dnes běžně využívají v profesionálních sférách. Koupě a použití zmíněných softwarů snižuje náklady a čas na produkci. Procedurální techniky jsou na vzestupu i díky neustélému růstu výkonu a paralerizace hardwaru.

V dnešní době je pod pojmem procedurální techniky myšlena zejména snaha napodobit přírodní útvary. Úspěšně lze napodobit nejrůznější materiály a tělesa, od vody a ohně až po skály a kopce [3].

Proces tvorby objektů podobných realitě je značně časově náročný. Za předpokladu, že i realita se řídí jistými algoritmy, lze tyto algoritmy napodobit a dosáhnout tak reálnějšího vzhledu v kratším čase. Například ručně vymodelovat detailní sněhovou vločku je zdlouhavá operace. Můžeme ovšem pro vymodelování sněhové vločky použít procedurální generování, v tomto případě zabere čas pouze výběr správných parametrů. Jinými slovy, procedurální generování je vytvoření požadovaného obsahu pomocí algoritmů. Obecně se při procedurálním generování využívá malého množství algoritmů, požadovaného efektu se obvykle dosahuje jejich vícenásobným skládáním s rozdílnými parametry.

V této práci se seznámíme se základy procedurálního generování zaměřené především na terén. Popíšeme několik používaných algoritmů a realizaci demonstrační aplikace.

2 Reprezentace Terénu

Ať už se jedná o vizualizaci měst, objektů, či animací, je jejich součástí obvykle terén. Terén samotný lze vykreslit i uložit mnoha způsoby. Budeme se zabývat zejména abstraktním terénem, jenž je složen z pseudonáhodných dat. Pokud bychom chtěli vykreslit fixní data, například mapy, procedurální generace by nebyla moc užitečná. Dala by se použít na detaily, které z map nejsou znatelné, například generování trávy.

Reprezentací je zde myšlena definice a formát dat v RAM a GPU. Uložení na HDD lze realizovat pomocí řady formátů (.obj, .dae, .3ds, .ply, atd.).

Pro popis objektů je možno použít spousty typů struktur. V demonstrační aplikace je využíván Mesh a Výšková mapa.

2.1 Mesh

Mesh reprezentuje povrch tělesa v počítačové grafice. Je uložen pomocí pole vrcholů a stran, kde každých N (N je minimálně 3) vrcholů představuje jednu stranu. Spolu s vrcholy lze ukládat i jiná data, například se běžně ukládají normály, texturovací souřadnice a tangenty. Aby se zabránilo duplicitě a plýtvání paměti, můžeme některé vrcholy pomocí indexů využít vícekrát. Při použití indexů je povrch tělesa definován vrcholy a indexy, kde každých N indexů odkazuje na N vrcholů. Znamená to tedy, že více indexů může použít stejný vrchol. Pokud nejsou vrcholy využity vícekrát, je použití indexů zbytečné.

Terén je ve své podstatě 3D těleso, proto jej lze uložit stejně jako všechny jiné 3D objekty. Mesh je tedy jednou z voleb pro ukládání a zobrazování terénu.

Pokud chceme mesh zobrazit, je třeba jej nějakým způsobem poslat na GPU. Mesh lze do GPU posílat před každým snímkem nebo pouze jednou.

2.1.1 Posílání meshe před každým snímkem

V dřívějších dobách, kdy GPU jestě neměly dostatečnou paměť, bylo nutné posílat na GPU celý mesh před každým vykreslením.

Je to jednodušší metoda, která je v dnešní době použitelná pouze v případě, že nepotřebujeme vykreslit mnoho objektů, anebo v případě, že data, jež vykreslujeme, se neustále mění.

2.1.2 Poslání meshe jednou, Vertex Buffer Object

Mesh je nejdříve jednou poslán do speciálního bufferu na GPU a poté pomocí tohoto bufferu opakovaně vykreslován. Mesh se na GPU ukládá do speciálního bufferu nazývaného Vertex Buffer Object, zkráceně VBO. Při vytváření VBO nám GPU vrátí identifikátor, který je použit při vykreslení.

Vykreslení se provede pouze dvěma příkazy:

- glBindVertexArray(vertexBufferObject); Nabindování pomocí identifikátoru, čímž řekneme GPU, že chceme použít konkrétní VBO.
- glDrawElements(GL_TRIANGLES, verticesNumber, GL_UNSIGNED_INT, 0); Samotné vykreslení z VBO.

VBO lze považovat za optimalizaci, neboť data, která se nemění, jsou poslána pouze jednou. Tato optimalizace je podporována až od OpenGL verze 1.4.

2.2 Výšková mapa

Výšková mapa je sada bodů, které reprezentují výšku terénu. Velmi často se popisuje pomocí textury, jejíž pixely zde reprezentují výšky jednotlivých bodů. Výšku lze uložit například do jednoho kanálu, což znamená, že jsme limitováni pouze na 255 hodnot. Vzniká tedy problém ztráty přesnosti. Z tohoto důvodu se někdy výška kóduje přes všechny kanály. V takovém případě máme až 16 581 375 hodnot pro texturu bez alpha kanálu nebo až 4 228 250 625 hodnot pro texturu ze všemi čtyřmi kanály. Samozřejmě lze použít i jine formáty než jen 8RGB, takovéto formáty jsou ovšem hardwarově podporovány jen na novějších platformách.

Zde je ukázka funkcí pro zakódování a dekódování desetinného čísla do a z barvy o třech kanálech [2]. Lze takto zjistit cílovou výšku z textury a posunout o ní konkrétní vertex přímo ve vertex shaderu.

```
vec3 packFloat(float f) { // float to rgb color
  vec3 color;
  color.b = floor(f / 256.0 / 256.0);
  color.g = floor((f - color.b * 256.0 * 256.0) / 256.0);
  color.r = floor(f - color.b * 256.0 * 256.0 - color.g * 256.0);
  return color / 256.0;
}
float unpackFloat(vec3 color) { // rgb color to float
```

```
    return color.r + color.g * 256.0 + color.b * 256.0 * 256.0;
}
```

V případě použití textury se výška ukládá do mřížky, která je definována šířkou a výškou textury, což lze považovat za nevýhodu. Výšková mapa se někdy nazývá jako 2.5D objekt, neboť se nejedná o plnohodnotný 3D objekt. Nabízí ovšem více možností než 2D. Výšková mapa potřebuje pro vykreslení přizpůsobený shader. I přesto je výšková mapa značně využívána, neboť oproti mesh reprezentaci šetří místo.

2.3 Objemová reprezentace

Trojdimenzionální data lze uložit jako objemovou reprezentaci, například do voxelové mřížky. V případě voxelové mřížky se jedná v podstatě o několik na sebe navrstvených 2D textur. Voxelová mřížka má tedy podobné nevýhody jako výšková mapa: Data jsou uložena v pevně dané mřížce závislé na rozlišení a velikosti voxelové mřížky. Jedná se o plnohodnotné 3D těleso, lze je tedy použít pro tvorbu terénu s převisy nebo jeskyněmi. Odstraňuje topologické nedostatky výškové mapy. Proto se objemová reprezentace používá jako doplněk k výškové mapě, výšková mapa je použita jako základ, a pokud chceme přidat jeskyně, využijeme objemovou reprezentaci [12].

Pro převod objemové reprezentace na povrchovou reprezentaci se používají variace algoritmu Marching Cubes [8] či Transvoxel [9].

Objemová reprezentace je užitečným formátem, jenž se dá využit pro vykreslování terénu. V demonstrační aplikaci je ovšem využit jen mesh a výšková mapa, neboť tyto dvě reprezentace dostatečne splňují požadavky. V budoucí práci by bylo ideální pomocí objemové reprezentace vykreslit i jeskyně či převisy.

3 Procedurální generování

Procedurální generování spočívá v algoritmickém generování požadovaného obsahu. Což znamená, že místo manuálního modelování, se objekt vytváří pomocí řady naprogramovaných algoritmů.

3.1 Výhody

Jednou ze základních výhod je, že při modelování lze dosáhnout reálnějších útvarů v relativně kratším čase.

Často zmiňovanou výhodou je také abstrakce, což znamená, že vlastnosti vygenerovaného útvaru jsou definovány několika parametry. Útvar lze kontrolovat parametricky. Stačí tedy uložit pouze těchto několik parametrů namísto vygenerovaného tělesa.

V případě praktických aplikací lze zvolit libovolnou vzorkovací frekvenci nebo jí upravovat podle vzdálenosti ke kameře.

Je si také třeba uvědomit, že jakákoliv generace je založena na náhodných číslech. A náhodná čísla samotná jsou založena na fyzikálních principech nebo na matematických algoritmech, což znamená, že náhodná čísla vlastně nejsou náhodná. Proto se setkáváme s pojmem pseoudonáhodnost, což tuto skutečnost zvýrazňuje. Výhodou je, že čísla takto generována lze předpokládat a opakovaně vypočítat. Je tedy možné vygenerovat pokaždé naprosto identické tělesa.

3.2 Nevýhody

Mezi nevýhody patří zejména nemožnost lokálních úprav, neboť celé těleso je upravováno pouze za pomocí několika parametrů. Proto se obvykle útvar vygeneruje a uloží do podoby vrcholů a hran, a následné úpravy se provádí již nad vygenerovaným tělesem. Nevýhodou tohoto postupu je, že velikost uloženého vygenerovaného tělesa může značne narůst.

Použité algoritmy nejsou samy o sobě náročné, ovšem požadovaného výsledku se dosahuje jejich několikanásobným skládáním. Výsledku se může dosáhnout za pomocí tisíce instancí algoritmů s nejrůznějšími parametry. V takovémto případě se výpočetní časy jednotlivých instancí algoritmů sčítají, čímž může celkový výpočetní čas značně narůst.

3.3 Algoritmy

Existuje nespočet algoritmů a jejich variací pro procedurální generování. V jejich názvech i popisech se někdy vyskytuje slovo šum (noise). Šumem je obyvkle myšlena funkce s následujícími vlastnostmi:

- Malá změna vstupní hodnoty způsobí malou změnu výstupní hodnoty (koherence).
- Velká změna vstupní hodnoty způsobí náhodnou změnu výstupní hodnoty.
- Šum má známý rozsah (obvykle od 0 do 1).
- Nevykazuje, nebo se snaží vyhnout očividné periodicitě. Bohužel každá funkce je periodická, proto je snaha o co nejdelší periodu.

3.3.1 Perlin noise

Perlin noise je někdy také nazýván Classic noise. Jedná se o šumovou funkci,kterou v roce 1982 představil profesor Ken Perlin. Byla využita při produkci filmu Tron, díky čemuž za ní Ken Perlin dostal v roce 1997 cenu Technical Achievement Award 3.

Perlin noise je souvislý šum, to znamená, že hodnoty jsou spojité. Jednoduše si jej lze představit jako variaci interpolace použité při čtení z textury. Základem je pole, jenž obsahuje náhodně vygenerované gradienty, neboli směry. Tyto gradienty jsou buď vygenerovány jednou při startu aplikace, anebo napevno zakompilovány jako pole konstant.

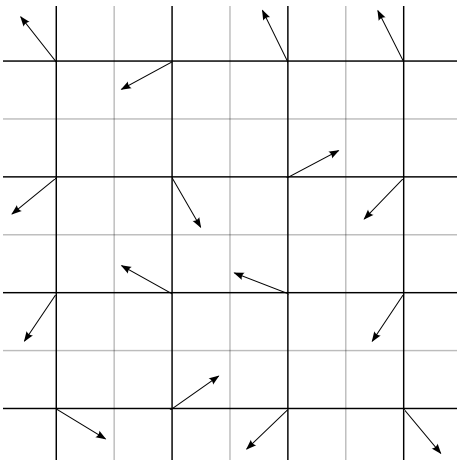
```
gradients = [ vec2( cos(random()), sin(random()) ), ... ]
```

Vyberou se čtyři nejbližší gradienty a pomocí nich se vypočítá výsledná hodnota. Výběr gradientů je lepší realizovat pomocí zahashované pozice [10], tak lze docílit větší variace s menším polem gradientů.

```
uint hash(uint x, uint y) {
  return (uint) ((( OFFSET_BASIS ^ (uint)x) * FNV_PRIME) ^ (uint)y) * FNV_PRIME);
}
```

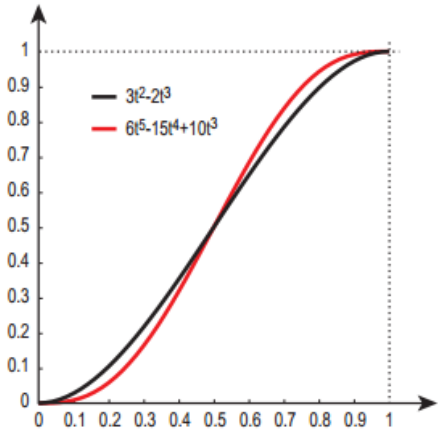
```
float perlin2D(float x, float y) {

  gradient_1 = gradients [ hash( uint(x)   , uint(y)   ) % gradients.length ];
  gradient_2 = gradients [ hash( uint(x)+1 , uint(y)   ) % gradients.length ];
  gradient_3 = gradients [ hash( uint(x)   , uint(y)+1 ) % gradients.length ];
  gradient_4 = gradients [ hash( uint(x)+1 , uint(y)+1 ) % gradients.length ];
```

```
    direction_1 = vec2(x,y) - vec2(int(x)   , int(y)   );
    direction_2 = vec2(x,y) - vec2(int(x)+1, int(y)   );
    direction_3 = vec2(x,y) - vec2(int(x)   , int(y)+1);
    direction_4 = vec2(x,y) - vec2(int(x)+1, int(y)+1);

    dot_1 = dot( gradient_1, direction_1 );
    dot_2 = dot( gradient_2, direction_2 );
    dot_3 = dot( gradient_3, direction_3 );
    dot_4 = dot( gradient_4, direction_4 );

    float S(float x) {
      return 3*x*x - 2*x*x*x;
    }
    a = dot_1 + f(dot_2 - dot_1);
    b = dot_3 + f(dot_4 - dot_3);

    return a + f(a - b);
}
```



Obrázek 1: Příklad gradientů dvojrozměrného Perlin noise.

Použitá interpolační křivka $S(t) = 3t^2 - 2t^3 \;\; t \in [0, 1]$ je jedním ze stavebních kamenů, jenž zajišťuje spojitost, neboť má spojitou druhou derivaci zleva i zprava. Později byla změněna na $S(t) = 6t^5 - 15t^4 + 10t^3 \;\; t \in [0, 1]$ neboť tato křivka má nulovou druhou derivaci v t=0 a t=1. Spojitost druhé derivace u této křivky zajišťuje plynulý přechod gradientů.



Obrázek 2: Rozdíl mezi starou a novou interpolační křivkou [5].

Uvedený algoritmus je jednodušší pro pochopení, existují ovšem i optimalizovanější varianty. Lze si jednoduše odvodit výpočet Perlin Noise i v ostatních dimenzích. Pro lepší vzhled se míchá několik oktáv perlin noise, toto míchání se někdy nazývá Fractal noise.

```
for(int i=0; i<octaves; i++) {
        result += perlinNoise2D(x*frequency, y*frequency)*height;
        height /= persistance;
        frequency *= persistance;
}
```

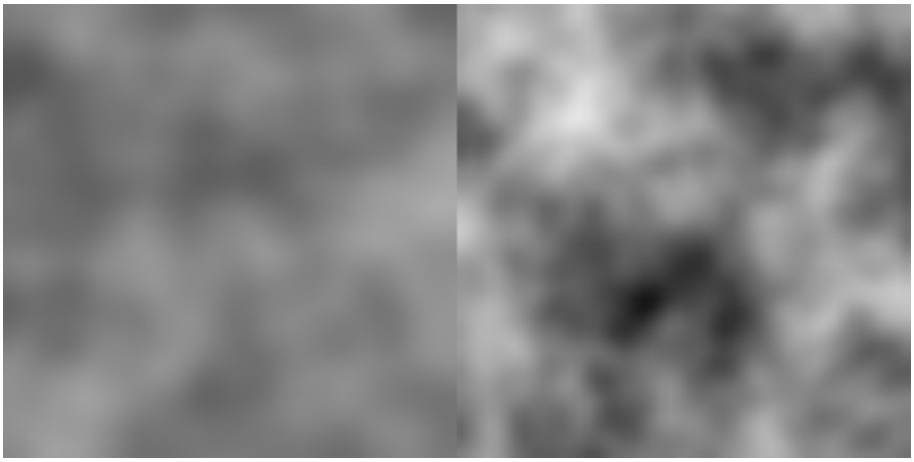
Výhodou perlin noise je jeho rozšířenost a známost [5].

3.3.2 Simplex noise

Ačkoliv je Perlin noise častěji zmiňovaný, obsahuje několik nedostatků. Tyto nedostatky se Ken Perlin rozhodl v roce 2001 odstranit uvedením nového šumu nazvaného Simplex noise. Simplex noise oproti Perlin noise obsahuje několik vylepšení:

- Simplex noise je celkově rychlejší a má méně operací s násobením, je tedy snadněji implementovatelný jak softwarově, tak i hardwarově.
- Simplex noise má ve vyšších dimenzích lepší výpočetní složitost: Simplex noise $O(n^2)$, Perlin noise $O(2^N)$, kde n = počet dimenzí. Jelikož lidem je nejbližší 2D a 3D dimenze, rozdíl ve složitosti mezi Perlin a Simplex noise v těchto dimenzích není až tak znatelný.

- Simplex noise má méně směrových artefaktů. Neboť u 2D Simplex noise jsou gradienty rozmístěny v trojúhelníkové mřížce, jsou zde tedy tři směry oproti pouze dvou směrům u 2D Perlin noise.
- Simplex noise častěji dosahuje krajních hodnot. Perlin noise se ke krajním hodnotám pouze přibližuje.



Obrázek 3: Vlevo Perlin noise, vpravo Simplex noise.

3.3.3 Worley noise

Worley noise je šumová funkce, kterou představil pan Steven Worley v roce 1996. Někdy je také nazýván Vornoi noise (kvůli podobnosti s Voroného diagramem), nebo cell noise (podobá se buňkám, cell je anglicky buňka). Vyobrazuje vzdálenost k nejbližšímu bodu nebo bodům. Nejjednodušší implementace je nejprve jednou naplnit pole náhodnými body.

```
positions = [ vec2( random(), random()) , ... ]
```

Poté pro náš bod najít nejbližší bod z pole a vzdálenost k němu. A nakonec tuto vzdálenost zobrazit jako barvu.

```
float WorleyNoise(vec2 myPosition) {
  float closestDistance;
```

[0]Obrázky v této sekci byly vygenerovány za pomocí aplikace dostupné na `http://aftbit.com/cell-noise-2/`

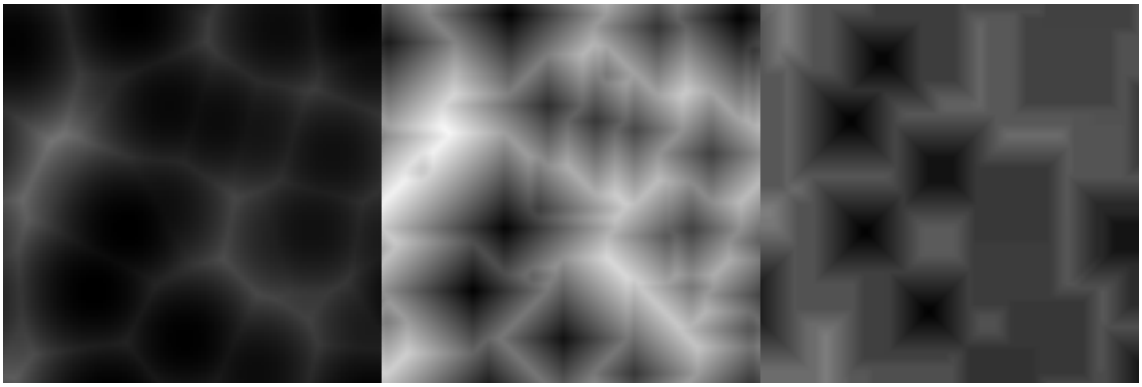
```
    foreach(position in positions) {
        float distance = Distance(myPosition, position);
        if (distance < closestDistance) {
            closestDistance = distance;
        }
    }
    return closestDistance;
}
```

Tento přístup je ovšem v závislosti na počtu bodů náročný na výpočetní čas. Těžší, ovšem efektivnější metoda je rozdělit plochu do síti čtverců a vygenerovat body pouze ve čtvercích nejbližších našemu bodu. Čili zahešovaná pozice našeho bodu bude použita jako semínko náhodného generátoru pozic. Dále lze použít několik funkcí pro výpočet vzdáleností, jejich rozdíl je viditelný na obrázku [4].

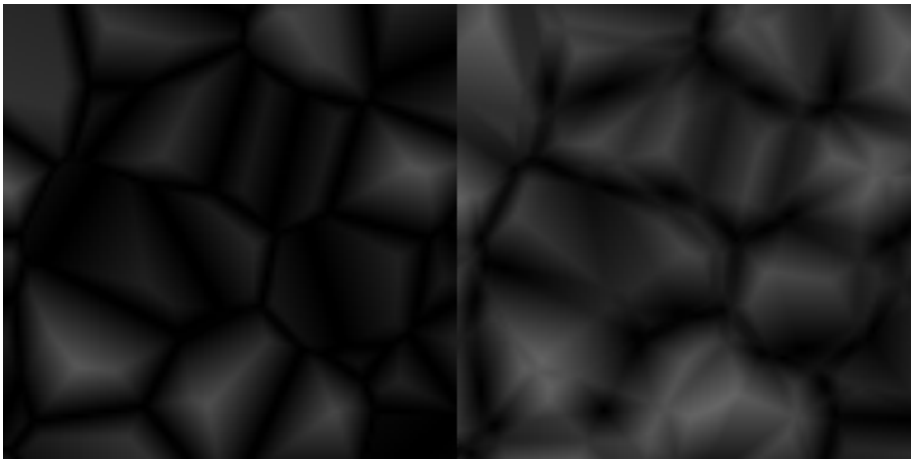
```
float EuclidianDistance(vec2 a, vec2 b) {
    float x = a.x - b.x;
    float y = a.y - b.y;
    return x*x + y*y;
}
float ManhattanDistance(vec2 a, vec2 b) {
    return abs(a.x - b.x) + abs(a.y - b.y);
}
float ChebyshevDistance(vec2 a, vec2 b) {
    vec3 diff = a - b;
    return max(abs(diff.x), abs(diff.y));
}
```

Goniometrické funkce a odmocnina jsou velmi výpočetně náročné, proto se jim snažíme co nejvíce vyhnout. Všimněte si, že při výpočtu Euclidovské vzdálenosti není použita odmocnina (Euclidian Distance: $(a, b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2}$). Tudíž všechny vzdálenosti, se kterými pracujeme, jsou umocněny na druhou.

Někdy se pracuje i s více vzdálenostmi, obvykle se jako D0 značí nejbližší vzdálenost a D1 jako druhá nejbližší vzdálenost, a tak dále. Výsledná zobrazená vzdálenost je potom jejich rozdíl, příklady jsou na obrázku [5].



Obrázek 4: Příklad Euclidian, Manhattan a Chebyshev Distance D0.



Obrázek 5: Příklad Euclidian Distance D1-D0 a D2-D0.

3.3.4 Diamond-square algoritmus

Tento algoritmus poprvé představili pánové Fournier, Fussell a Carpenter na konferenci SIGGRAPH v roce 1982. Někdy se nazývá jako Plasma, či Cloud fractál, neboť připomíná plasmu či oblaka.

Pro 2D dimenzi nejprve algoritmus vygeneruje čtyři základní rohové body. Následně se spustí iterativní algoritmus, který se dá rozdělit do několika kroků:

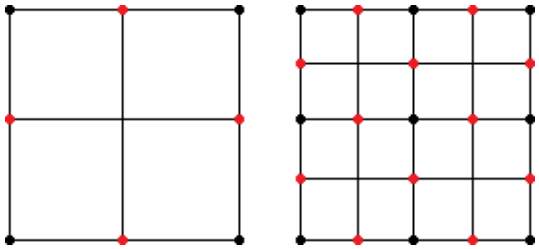
- Pro každou hranu čtverce se spočítá průměr dvou bodů definujících jeho hranu. Tento průměr, plus náhodný offset, je nová hodnota bodu mezi těmito dvěma body.

[0]Některé obrázky v této sekci byly vygenerovány za pomocí aplikace dostupné na `http://www.somethinghitme.com/projects/canvasterrain/`

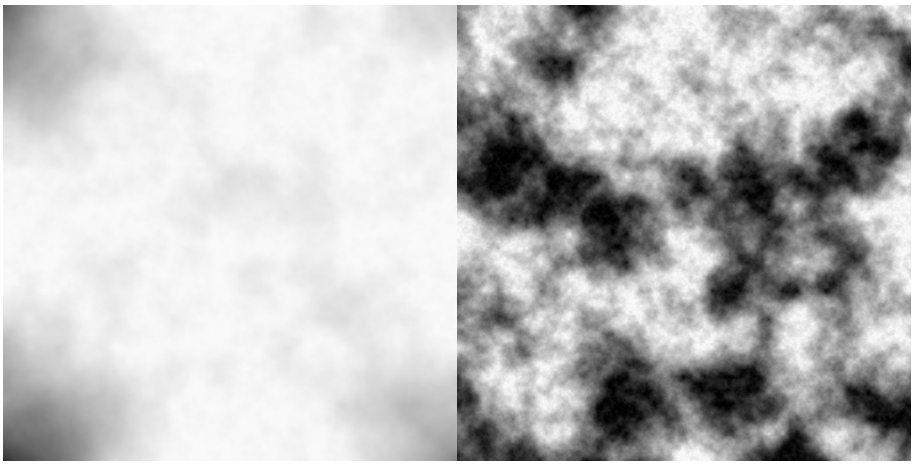
- Spočítá se průměr čtyř právě spočítaných bodů, plus náhodný offset, a uloží se do bodu uprostřed právě počítaného čtverce.
- Vzniklo pět nových bodů, které nám rozdělily čtverec na další menší čtyři čtverce. Na každý nově vzniklý čtverec se spustí další iterace algoritmu.

Zmiňovaný offset je obvykle náhodný a jeho rozsah určuje hladkost výsledku. Rozsah se s každou iterací zmenšuje. Tento rozsah se obvykle nazývá roughness.

Z algoritmu je zřejmé, že velikost výsledných dat v jedné ose je mocnina na druhou plus jedna. Tento algoritmus bohužel nebyl využit, neboť není vhodný pro generaci nekonečných dat.



Obrázek 6: Kroky konstrukce při použití Midpoint displacement.



Obrázek 7: V levo Midpoint displacement z roughness 1 a vpravo z roughness 10.

4 Popis realizace demonstrační aplikace

Cílem demonstrační aplikace bylo naprogramovat nekonečný průchozí terén, jenž je generován pomocí několika výše zmíněných algoritmů. Terén je rozdělen do sítě čtverců o fixní velikosti, kde každý čtverec je ukládán jako mesh do VBO. Generují se pouze čtverce nacházející se v nastavené vzdálenosti od kamery. Většinu parametrů lze měnit v xml souboru umístěném v Release/Configuration.xml, lze také vypnout či zapnout jednotlivé algoritmy pomocí use parametrů. Pokud jsou zapnuté oba algoritmy, je výsledná výška terénu jejich součet. Při vypnutí obou algoritmů bude terén pouze rovná plocha, i v tomoto případě je terén tessallován, výška nových tessalovaných bodů je upravena o několik oktáv Perlin noise textury.

4.1 Použité technologie

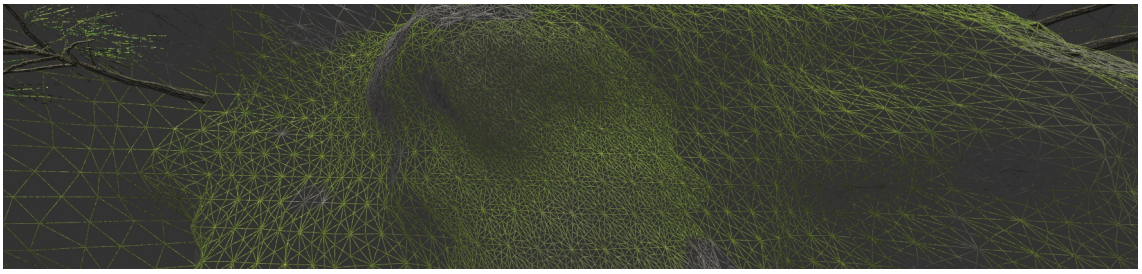
Demonstrační aplikace je naprogramovaná v C++11 a Visual Studiu 2012, z nových vlastností C++11 využívá zejména klíčového slova auto a foreach cyklu. Je použita grafická knihovna OpenGL a její rozšíření ve formě GLEW a GLUT. Shadery jsou napsány v defaultním OpenGL shader jazyce, čili GLSL. Části architektury tříd jsou převzaty zejména z Unity3D [11], podobné vzory jsou použity i v ostatních komerčních enginech.

4.1.1 Tessellace

Je využita triangle tessellace, což znamená že tessellace se provádí nad trojúhelníky. Level tessellace jedné strany trojúhelníku je závislý na průměru vzdáleností dvou bodů strany od kamery. Level tessellace je také zaokrouhlen k nejbližší mocnině dvou, neboť plynulý přechod vypadá nepřirozeně. Takto se tessallační level pouze zdvojnásobí, což je viditelné na obrázku [8].

```
float closestPowerOf2(float a) {
    return pow(2, ceil(log(a)/log(2)));
}
float tessLevel(float d1, float d2) {
    float d;
    d=(d1+d2)/2;
    d=clamp((1/d)*TESS_DETAIL), 1, 64);
    d=closestPowerOf2(d);
    return d;
}
```

```
void TESSALLATION_SHADER() {
    // zkopírování dat každého vrcholu

    // spočítání vzdálenosti bodů trojúhelníku od kamery
    float d0=distance(i[0].worldPos,EyePosition);
    float d1=distance(i[1].worldPos,EyePosition);
    float d2=distance(i[2].worldPos,EyePosition);

   // zjištění tessallačního levelu
    gl_TessLevelOuter[0] = tessLevel(d1,d2);
    gl_TessLevelOuter[1] = tessLevel(d0,d2);
    gl_TessLevelOuter[2] = tessLevel(d0,d1);

    gl_TessLevelInner[0] = gl_TessLevelOuter[2];
}
```



Obrázek 8: Ukázka tessallace.

Pozice tessallovaných vertexů je poté upravena za pomocí několika oktáv perlin noise textury. Perlin noise lze generovat na grafické kartě. Pokud ovšem není zapotřebí vysoká variace a kvalita, lze použít i předgenerovanou navazující texturu. Při použití několika oktáv není horší kvalita předgenerované textury ani periodicita znatelná.

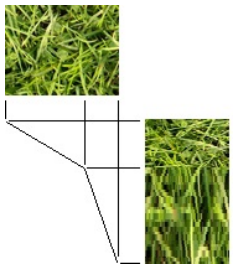
4.1.2 Triplanar texture projection

Jedním z problémů je mapování textury na terén. Nejjednodušší je namapovat texturu podle dvou souřadnic, například X a Z. Poté ale bude textura vypadat dobře pouze při pohledu shora, avšak při pohledu z boku bude nepřirozeně roztažená [9].

Dá se říci, že pokud normála přesáhne určitou hodnotu, bylo by lepší mapovat texturu podle jiných souřadnic. Přesně toto se děje při triplanar texture projection. Textura

se mapuje ze tří stran, podle YZ, ZX a XY. Výsledná barva je potom závislá na normále, váha každé strany je jedna složka normály.

```
vec3 triPlanar (texture, position, normal, scale) {
  position *= scale;
  vec3 blendWeights = normalize(abs(normal));
  vec4 color1 = texture2D(texFromSide, position.yz);
  vec4 color2 = texture2D(texFromTop, position.zx);
  vec4 color3 = texture2D(texFromSide, position.xy);
  return
    color1 * blendWeights.x +
    color2 * blendWeights.y +
    color3 * blendWeights.z ;
  }
```



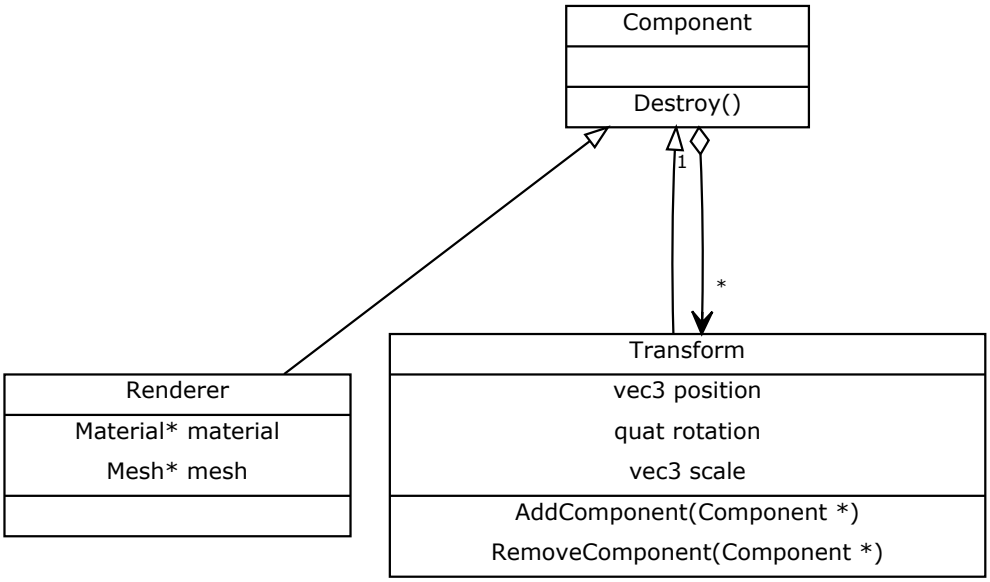
Obrázek 9: Pohled shora a ze strany na terén namapovaný pouze podle XZ.

4.1.3 Component a Transform design

Základem všeho je třída Component [10], z této třídy dědí ostatní třídy. Jakýkoliv počet Component lze poté přidávat do třídy Transform. Transform má kromě listu děti (Component) také pozici, rotaci a scale, z čehož se počítá modelová matice pro komponentu Renderer.

Tyto třídy lze poté poskládat například takto:

```
Transform
|−Renderer
|−Transform
| |−Renderer
| |−Renderer
```



Obrázek 10: Diagram tříd component a transform designu.

Tento design je využíván pro jednodušší práci ve scéně. Jeden čtverec terénu má svůj Renderer a poté několik dětí Transformů – stromů. Pro smazání čtverce, včetně všech jeho potomků, stačí zavolat metodu Destroy pouze na tento jeden čtverec.

4.1.4 Detail textures

Podíváme-li se z dálky či z blízka na texturu, jenž je namapována v pevném měřítku, je viditelná interpolace nebo opakování textury. A to i přes použití Antialiasingu čí Antistopic filteringu. Proto je mapování textur v demonstrační aplikaci závislé na vzdálenosti od kamery. Textura je namapována v rozdílných měřítcích a poté se mezi nimi podle vzdálenosti od kamery plynule interpoluje.

```
vec3 color = mix(
    triPlanar (snow, worldPos, normal, 0.1),  // pod vzdálenost 100 je měřítko 0.1
```

```
    triPlanar (snow, worldPos, normal, 0.01), // nad vzdálenost 200 je měřítko 0.01
    smoothstep(100, 200, distance) // plynule interpoluje mezi oběma měřítky
);
```

4.2 Optimalizace

Efektivita a výkon jsou zejména důležité u aplikací, které se musí překreslovat několikrát za sekundu. U takových aplikací znamená výběr špatného přístupu ztrátu drahocenného výpočetního času. Proto je při vývoji takových aplikací kladen důraz na neustálé zvyšování jejich efektivity a výkonu. Tato činnost se obecně nazývá optimalizace. Jinými slovy, optimalizace je procházení kódu a hledání algoritmů nebo přístupů, jenž by program zrychlily či zmenšily jeho paměťovou náročnost.

V průběhu vývoje měla aplikace stabilní FPS, po přidání stromů se toto číslo zmenšilo až na 12 FPS. Hlavně na tento podnět byla nalezena a implementována většina následujících optimalizací. Implementace všech těchto optimalizací byla časově i programově náročná, bylo nutno několikrát přepsat velkou část kódu.

Aplikace byla testována na třech počítačích s konfiguracemi uvedenými v tabulce [1]. Výsledky měření se mohou na jiných konfiguracích lišit.

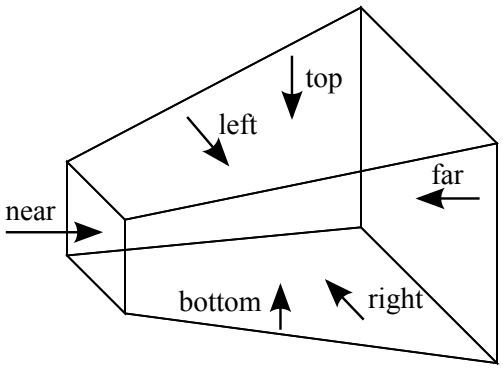
	PC1	PC2	PC3
CPU	FX-6300 VISHERA	i5-3570 @ 3.40Ghz	Core 2 Quad Q9650 @ 3.00GHz
GPU	GTX 760	Quadro 600	GTX 560 Ti

Tabulka 1: Konfigurace počítačů, na nichž byla aplikace testována.

4.2.1 View frustum culling

Někdy nazýván pouze frustum culling. Dá se volně přeložit jako vyřezání pohledové pyramidy. Jak název naznačuje, jedná se o vyřezání neboli vynechání všech instancí meshe mimo pohled kamery. Znamená to tedy, že instance meshe mimo pohled kamery se vůbec nevykreslují.

Základem této optimalizace je primitivum, v našem případě kvádr nazývaný bounding box, jenž obsahuje všechny vrcholy meshe. Ze součinu view a projection matice kamery se odvodí šest stěn, definující pohled kamery (view frustum [11]). Poté už se provede jen test, zdali bounding box každé instance meshe leží uvnitř těchto stěn.



Obrázek 11: View frustum - šest stěn, jenž ohraničuje pohled kamery.

Každá stěna pohledu kamery je reprezentována jako směr a vzdálenost ke středu souřadnicového systému.

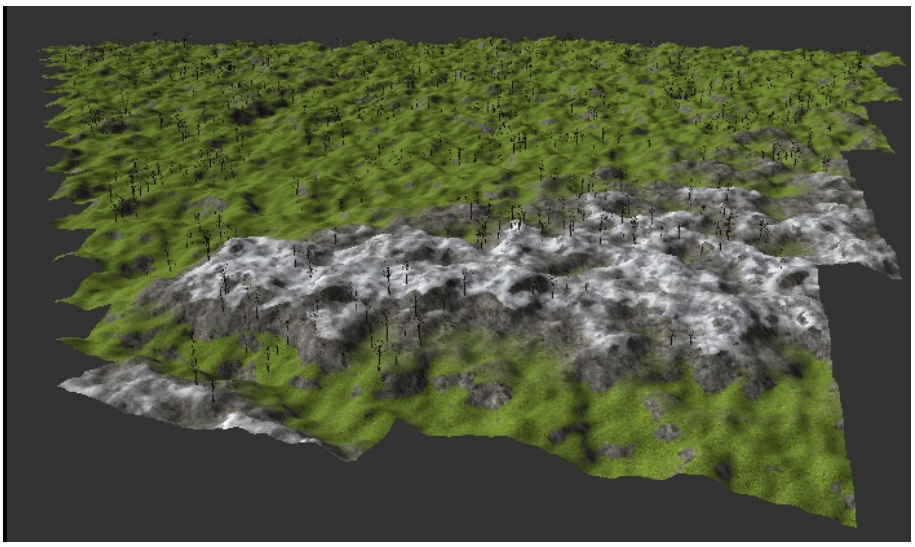
Bounding box se převede na kouli tak, že se vezme jeho střed a jeho nejdelší průsečík. Nakonec se zjistí vzdálenost středu koule ke všem stěnám pohledu kamery a pokud je tato vzdálenost menší než záporný poloměr koule, mesh není viditelný. Následuje pseudokód, jenž objasňuje implementaci view frustum cullingu.

```
float Plane::DistanceToPoint(vec3 point) {
  return dot(point,planeNormal)+planeDistance;
}

bool frustumCullingPassed=true;
for(uint i=0; i<6; i++) {
  if( viewPlanes[i].DistanceToPoint(instancePosition) < -boundingBox.GetRadius() ) {
    frustumCullingPassed=false;
    break;
  }
}

if (frustumCullingPassed) {
  // Pokud je mesh vyditelny, vykreslime jej
}
```

Z tabulky [2] je znatelné, že implementace měla značný vliv na FPS.

FPS	PC1	PC2	PC3
Vypnut view frustum culling	64	24	50
Zapnut view frustum culling	126	36	106

Tabulka 2: Vliv view frustum cullingu na FPS (bez stromů).



Obrázek 12: Ukázka pozastaveného view Frustum cullingu v kombinaci s Instanced renderingem.

4.2.2 Instanced rendering

Můžeme přeložit jako vykreslování instancí. Jedná se tedy o vykreslení všech instancí jednoho meshe najednou. Pod pojmem instance je myšlena modelová matice, která transfomuje mesh. Jinými slovy se všechny modelové matice pošlou do bufferu najednou, a poté se pouze třemi funkcemi všechny instance vykreslí.

Protože se zde pracuje především s instancemi, které nemění pozici, není potřeba posílat na GPU novou modelovou matici před každým vykreslením. Lze tedy vytvořit pole statických modelových matic, které se přesune na GPU do bufferu. Vykreslení všech instanci stejného meshe potom probíhá následovně:

- Nabindování meshe.
- Samotné vykreslení všech instanci meshe, kde instancesNum je počet modelových matic neboli počet instancí.

V kódu potom tento postup vypadá takto:

```
glBindVertexArray(VAO);
glDrawElementsInstancedBaseVertexBaseInstance(GL_TRIANGLES, verticesNumOnGPU,
    GL_UNSIGNED_INT, 0, instancesNum, 0, 0);
```

Hlavní podnět pro implementaci instanced rendering byl po přidání stromů. Neboť stromů se ve scéně nacházelo až několik tisíc a vykreslovat všechny jednotlivě si vyžádalo hodně výpočetního času. Z tabulky [3] je znatelné, že při použití instanced renderingu má počet vykreslených instancí opravdu menší vliv na poměrné výsledné FPS, ovšem pouze na grafických kartách pro něž má tato optimalizace smysl.

FPS bez stromů / ze stromy	PC1	PC2	PC3
Bez instanced rendering	54/27	13/12	68/33
Z instanced rendering	96/81	17/11	72/52

Tabulka 3: Vliv instanced renderingu na FPS, 1000 čtverců (bez frustum cullingu).

4.2.3 Porovnání konteineru vectoru a listu

Velkou roli hraje také samotná reprezentace dat v RAM paměti. Doba čtení souvislého kusu paměti je rychlejší, než čtení dat náhodně roztroušených.

List je dvojitě linkovaný list, což znamená, že každá položka obsahuje data a ukazatele do obou směrů, položky jsou tedy náhodně rozmístěny v paměti, což zvětšuje přístupovou dobu. Naopak vector je pouze souvislé pole dat, jenž je obsluhováno třídou, která se v případě rozšíření pole stará o jeho realokaci. U vectoru lze přistoupit přímo k poli dat pomocí metody data a využít na nich průchod pomocí ukazatele.

Volba konteineru je velmi důležitá, což dokazuje i fakt, že CryEngine [12] využívá STLport [13], což je optimalizovaná a efektivnější implementace STL.

4.2.4 Porovnání procházení pole pomocí iterátoru a pointeru

Ze začátku byl pro jednoduchost a přehlednost použit foreach cyklus:

```
for(auto shader : Shader::all) {
    shader->Use();
}
```

Tento foreach cyklus ovšem používá iterátor, lze jej tedy přepsat jako:

```
for(auto shader = Shader::all.begin(); shader != Shader::all.end(); ++shader) {
    shader->Use();
}
```

Bohužel iterátor využívá metody, které jsou náročnější než pouhé inkrementování ukazatele. Proto byly kontainery ve vykreslovací smyčce změněny na vector, což umožnilo průchod pomocí ukazatele:

```
shader = Shader::all.data();
shaderEnd = shader+Shader::all.size();
while(shader<shaderEnd) {
  shader->Use();
  shader++;
}
```

U jedné kritické smyčky tato změna přidala až 2 FPS.

4.2.5 Vykreslovací smyčka

Jedním ze stavebních kamenů programu, jenž vykresluje v realtime, je vykreslovací smyčka. Musí být naprogramována co nejefektivněji, neboť se spouští několikrát za sekundu. Většina výše zmíněných optimalizací se prováděla především ve vykreslovací smyčce.

Ze začátku vypadala smyčka následovně:

```
foreach object in all_objects
    bind object.material.shader
    bind object.material.shader.uniforms
    bind object.material.uniforms
    bind object.modelMatrix
    bind object.mesh
    draw
```

Pro každý objekt se znovu nabinduje vše potřebné. Pokud se používá pouze jeden shader, GPU ovladač to zjistí a nic neudělá (nechá tam již dříve nabindovaný shader). Pokud ale například použijeme dva shadery a objekty v paměti je střídají, hodně tím klesá FPS. Proto byla vykreslovací smyčka upravena na následující, velice podobnou smyčku, jenž využívají i komerční enginy.

```
foreach shader in all_shaders
  bind shader.uniforms
  foreach material in shader.materials
    bind material.uniforms
    foreach mesh in material.meshes
      bind mesh
      bind all model matrices
      draw instanced
```

V této smyčce se v podstatě každý shader, materiál, či mesh použije opravdu téměř pouze jednou. Z pseudo kódu je znatelné že:

- Každý shader si musí pamatovat materiály, které jej používají.
- Každý materiál si musí pamatovat meshe, které jej používají.
- Každý mesh si musí pamatovat všechny instance, které jej používají.

4.2.6 Level of Detail

Je zbytečné vykreslovat vzdálený objekt za pomocí tisíce vrcholů, když je ve výsledku viditelných pouze několik pixelů. Proto se využívá Level Of Detail systém, ze vzdálenosti se mění detail meshe.

Mesh má několik téměř stejných variant, které se liší hlavně v počtu vertexů. Varianty s menším počtem vertexů se zobrazí, když je objekt vzdálenější od kamery. Testovat vzdálenost od kamery každého z tisíce stromů každý snímek je teoreticky zbytečné. Proto se vzdálenost testuje pouze co desetinu sekundy.

Možnou optimalizací je provádět testování vzdálenosti a výběr správného meshe na GPU pomocí geometry shaderu.

4.2.7 Více vláknová aplikace

Načítání textur a generování terénu, jsou časově náročné a blokující operace, proto probíhají v samostatných vláknech. Samotný časově náročný výpočet se provádí do lokálních

proměnných, zkopírování do používaných globálních proměnných se provádí v sekci zabezpečené pomocí mutexů.

4.3 Nepřesnost čísel s plovoucí desetinnou čárkou

Jedním z problémů představuje také reprezentace čísel. Čím větší číslo je, tím větší část je použita pro reprezentaci čísel před desetinnou čárkou, to znamená, že desetinná čárka se postupně posouvá doprava, čímž zmenšuje prostor pro desetinná čísla samotné. Při navrhování matematických operací je potřeba tuto limitaci brát v potaz a vyvarovat se násobení nebo dělení velkými čísly. Pokud tohle nestačí, je nutno použít čísla s pevnou desetinnou čárkou nebo počítat větší vzdálenosti odděleně, například přesunout celý svět co nejblíž k nule každých sto metrů a tento posun si zaznamenat v jiné proměnné. Dají se takto vylepšit fyzikální výpočty. Tyto úpravy jsou ovšem náročné na implementaci, neboť ovlivňují všechny výpočty v aplikaci.

Pokud se v demonstrační aplikaci vzdálíme dostatečně daleko od středu, lze ve vygenerovaném terénu pozorovat jisté artefakty způsobené ztrátou přesností čísel.

5 Zhodnocení

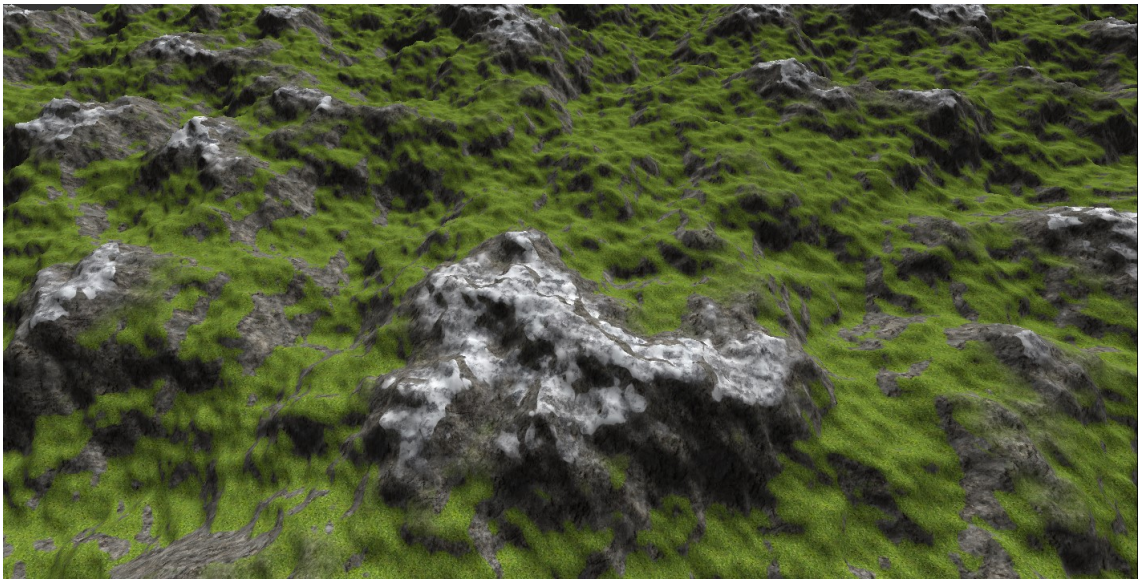
Z výsledku z tabulky [4] je znatelné, že generace pomocí Perlin noise dosahuje lepších časů. A to i přes použití deseti oktáv. I proto Perlin noise patří k oblíbenějším algoritmům. Simplex noise se většinou nepoužívá, neboť je méně známý a má v druhé dimenzi stejnou složitost. Pokud ovšem potřebujeme šum, jenž často dosahuje krajních hodnot (viditelné na obrázku [3]), použijeme Simplex noise.

Worley Noise, ačkoliv byl optimalizován, je pořád relativně pomalý. Používá se, neboť charakterisitka jeho výstupu je odlišná od ostatních šumových algoritmů. Jeho výstup více připomíná přirozené hory a kopce (viditelné na obrázku [14]).

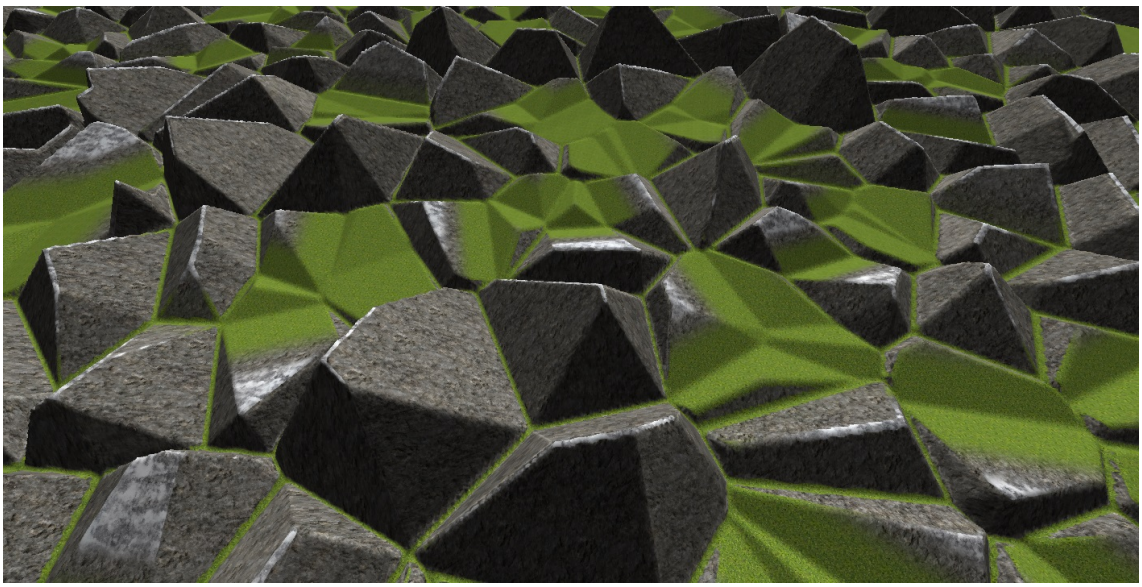
Je viditelné, že rychlost generace je závislá na rychlosti jednoho jádra. Je tomu tak, neboť generace samotná probíhá v samostatném vlákně na CPU. V budoucnu by bylo lepší generaci více paralerizovat nebo přesunout na GPU. Pokud je pohyb kamery příliš rychlý, je možné dostat se do situace, kdy není ještě vygenerován žádný viditelný terén.

Průměrný čas generování v sekundách	PC1	PC2	PC3
Perlin Noise	0.0026	0.0006	0.0019
Worley Noise	0.0697	0.0503	0.0716

Tabulka 4: Průměrná doba generování 3000 čtverců terénu v sekundách.



Obrázek 13: Ukázka vygenerovaného terénu pouze za pomocí Perlin noise algoritmu.



Obrázek 14: Ukázka vygenerovaného terénu pouze za pomocí Worley noise algoritmu.

6 Závěr

Výsledkem této práce je implementace algoritmů a optimalizací na praktickém příkladu aplikace pro generovaní nekonečného terénu. Uživatel si může vyzkoušet výhody a nevýhody obou algoritmů jakožto i vliv jednotlivých parametrů na výsledný terén.

Teoreticky jsme popsali řadu algoritmů z nichž jsme prakticky naiplementovali Perlin a Worley noise, Midpoint displacement nebyl použit, neboť se ukázal být nevhodný pro nekonečný terén. Worley noise je oproti Perlin noise časově náročnější, lze jim ovšem přesněji napodobit hornaté terény. Pro realtime vykreslování jsou důležité optimalizace, proto bylo použito několik optimalizačních algoritmů, z nichž nejlepší výsledky měl Frustum culling a Instanced rendering.

V rámci této práce se objevila spousta dalších možností urychlení a optimalizace. Proto bych rád v této práci pokračoval v rámci diplomové práce, kde bychom se zaměřili na paralerizaci, rychlejší výpočet a hlavně realističtější výsledky.

7 Reference

[1] KHRONOS, Group. *Official OpenGL website* [online]. 2014 [cit. 2014-4-21] . Dostupné z: http://www.opengl.org/

[2] STACK OVERFLOW. [online]. 2014 [cit. 2014-4-21] . Dostupné z: http://www.opengl.org/

[3] PERLIN, Ken. *Osobní stránky* [online]. [cit. 2014-4-21] . Dostupné z: http://mrl.nyu.edu/~perlin/

[4] ZUCKERE, Matt. *Perlin Noise Math FAQ* [online]. [cit. 2014-4-21] . Dostupné z: http://webstaff.itn.liu.se/~stegu/TNM022-2005/perlinnoiselinks/perlin-noise-math-faq.html

[5] GUSTAVSON, Stefan. *Simplex noise demystified* [online]. 2005 [cit. 2014-4-21] . Dostupné z: http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf

[6] ROSÉN, Carl-Johan. *Cell Noise and Processing* [online]. [cit. 2014-4-21] . Dostupné z: http://www.carljohanrosen.com/share/CellNoiseAndProcessing.pdf

[7] NVIDIA. *GPU gems* [online]. 2004 [cit. 2014-4-21] . Dostupné z: http://http.developer.nvidia.com/GPUGems/gpugems_part01.html

[8] CHERNYAEV, Evgeni V.. *Marching Cubes 33: Construction of Topologically Correct Isosurfaces* [online]. 1995 [cit. 2014-4-21] . Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.7139

[9] LENGYEL, Eric. *The Transvoxel^{TM} Algorithm* [online]. 2010 [cit. 2014-4-21] . Dostupné z: http://www.terathon.com/voxels

[10] LANDON, Curt Noll. *FNV hash* [online]. [cit. 2014-4-21] . Dostupné z: http://isthe.com/chongo/tech/comp/fnv/

[11] UNITY, Technologies. *Unity Game Engine* [online]. 2014 [cit. 2014-4-21] . Dostupné z: https://unity3d.com

[12] CRYTEK, GmbH. *CryEngine* [online]. 2014 [cit. 2014-4-21] .
Dostupné z: http://cryengine.com

[13] FOMITCHEV, Boris. *STLport* [online]. 2008 [cit. 2014-4-21] .
Dostupné z: http://stlport.sourceforge.net

[14] PLANETSIDE, Software. *Terragen* [online]. 2014 [cit. 2014-4-21] .
Dostupné z: http://planetside.co.uk

[15] E-ON, Software. *E-on Vue* [online]. 2014 [cit. 2014-4-21] .
Dostupné z: http://www.e-onsoftware.com

[16] WORLD MACHINE, Software, LLC. *WorldMachine* [online]. 2014 [cit. 2014-4-21] .
Dostupné z: http://www.world-machine.com

A Příloha

Součástí priloženého CD je:

- Elektronická verze této práce ve formátu PDF.
- Zdrojové kódy demonstrační aplikace.
- Spustitelný soubor demonstrační aplikace.